A Programmatic Approach to WWW Authoring using Functional Programming

Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
normark@cs.auc.dk

August 9, 2001

Abstract

A programmatic approach to authoring of static WWW documents is introduced. The main conclusion of the paper is that functional programming fits well with the needs of in the area of static WWW authoring. At the more detailed level it is concluded that the flexibility of a Lisp programming language is a good basis for a programmatic WWW author. The paper describes the interaction between programming and markup in the creation of static WWW pages. It is illustrated how the Scheme-based software package, LAML, can be used to support a flexible programmatic authoring process. As a key idea, LAML mirrors every element in HTML as a Scheme function. On top of the mirror, LAML offers a variety of document styles, tools, and other kinds of support.

1 Introduction

Authoring of World Wide Web material involves construction of documents with XML or HTML markup as well as some amount of server and client side programming. In this paper we will focus on the use of programmatic means for authoring of relatively static WWW documents.

In order to achieve a more precise discussion of static and dynamic WWW documents we will distinguish between four classes of documents, and four different binding times. The four document classes and the related binding times are illustrated in figure 1. The *binding time* represents the execution time of the involved program (if any); The WEB page is frozen to a fixed appearance at document binding time. In the one extreme, *static documents* are written directly in HTML, and no programming dynamics is involved at all. Such document are bound at edit time. In the other extreme, a *dynamic document* is never frozen, because program execution takes place in the browser at the time the document is being read. *Calculated documents* are frozen at the time the document is delivered by the WWW server. As such, they represent an important class of documents that are generated by programs running at

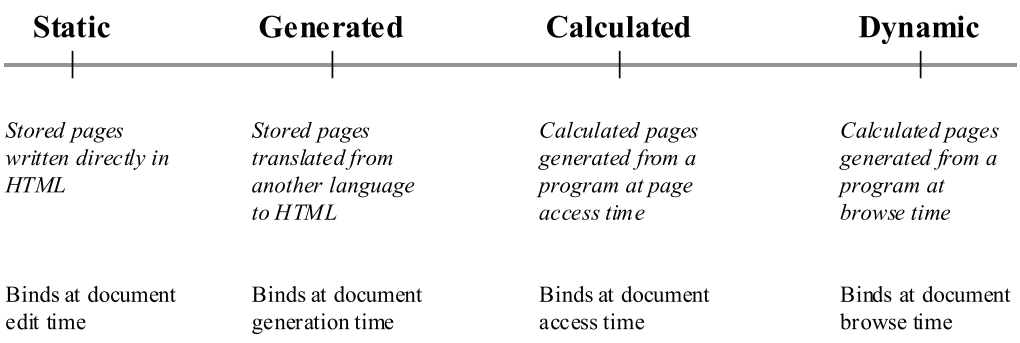

Figure 1: *Four different categories of WWW documents.*

the server. Finally, *generated documents* are bound at an earlier point in time, for instance by application of a transformation program that generates fixed HTML markup from higher level markup (in XML or a similar language).

In this paper we will focus on generated pages. More precisely, we will deal with authoring of Web documents using a programming language as the *document source language*. We reserve the term *programmatic authoring* for this endeavor. We will discuss the generation of WWW documents (in HTML) from a *document source program* (in a programming language). Thus, we are concerned with the situation where WWW documents are written by use of a programming language instead of a markup language. This calls for a number of considerations of the interaction between the programming language and the underlying and applied WWW technologies. This paper will present a possible solution to these challenges using a functional programming language.

Given the idea of a document source program for the creation of static WWW pages, several interpretations of the document source program may be possible. However, the most prominent of these seems to be the one that translates the document source program to HTML. Therefore, in our work, the execution of the document source program derives an HTML file, or a set of such files.

The most novel idea in our work is indeed the use of a programming language as a document source language. As it will be discussed below, some programming paradigms and languages are better suited than others in a programmatic authoring process. In our work we rely on the the functional programming paradigm and Scheme [4] - a language in the Lisp family.

It is a well-known fact that development of most non-trivial WWW documents involves some degree of programming. Many WEB documents are calculated at the server side as a response to input from the user. Other WEB documents are made by a tool, which in this context can be seen as a fixed program that generates the underlying HTML document from a higher level description. Yet other programs are used in dynamic documents at the client side (java applets, java script programs, and others). In our work we apply a programming language as authoring language for static WWW pages. The key

observation is that such a programmatic authoring approach creates an ideal ground for an integration of WWW technologies and the necessary elements of programming.

As one of our main elements, we mirror the HTML markup language in the programming language. This mirror allows us to work exclusively in the programming language without use of mixed language documents (such as an HTML shell with embedded program fragments). The uniform use of a single linguistic framework—in our work, a programmatic framework—is more powerful than a mixed approach. Mixing markup and program fragments in a single document creates borderlines between two linguistic universes which cannot smoothly interact with each other. In comparison between a markup language and a programming language, the latter is clearly the one holding most power. Thus, we go for an inclusion of the markup language in the programming language. The other possibility (extending the markup language with programming capabilities) is less attractive, as demonstrated by the embedding of Lisp in XML by the language called XEXPR [6].

In this paper we will argue that authoring of complicated WWW materials involves almost the same challenges - problems and solutions - as development of non-trivial software. This includes, for instance, general mastering of complexity, avoidance of redundancy, abstraction to eliminate distracting details, and modularization and separation of concerns. Using a programmatic approach to WWW authoring makes it natural and straightforward to apply a variety of programming techniques on WWW documents.

We are well aware that a programmatic approach to WWW authoring—brought to the extreme—will have a hard time to be successful for WWW authoring in the large. We believe, however, that there exists some niches in which the techniques proposed in this paper can play a significant role, both with respect to productivity and quality of the authored documents and WEB sites. Our own use of the LAML software packages gives us a clear indication in this direction.

In section 2 we discuss the general benefits of a programmatic WWW authoring approach. In section 3 the approach is illustrated by means of a concrete example. This gives a foundation for the next two sections, in which we discuss programming paradigm and programming language issues related to WWW authoring, and the LAML approach respectively. Related work will be described in section 6.

2 The benefits of a programmatic approach

Basically, programming is about *automating* a solution to a given problem. A programmed solution usually takes responsibility for the execution of a large number of steps—or evaluation of a complex expression—which otherwise should be done manually.

Equally important, programming is about *abstraction*. It is difficult and error prone to deal with a great number of primitive steps, or an expression with a large number of operands. The abstraction mechanisms supported by most

programming languages allow the programmer to formulate the program at a higher level. Some of the original steps or operands are encapsulated, named and generalized by means of parameters such that the resulting procedure or function calls involve fewer details.

It is our hypothesis that WWW authoring will benefit from the programmatic techniques mentioned above. Automation in the context of WWW authoring can be brought in by a variety of different means. Typically, however, the author applies a tool on a document in order to check if it fulfills some given properties, or in order to convert it to another format. If the source of the WWW document is a program, automated solutions can be integrated much more smoothly. As part of the document we can write a piece of program that carries out the needed calculation (check or transformation), or we can apply functionality from an existing library if it is available.

Abstraction is a key idea in XML as well as a much wanted (but missing) concept in HTML. But XML offers no computational dynamics, and as such it depends on external mechanisms such as XSL [2] for transformation purposes. If we instead use a programming language for authoring of WWW documents we can use the existing abstraction mechanisms of the programming language, as well as the inherent computational power of the programming language to achieve a lot of advantages. Using such a programmatic approach guaranties good, general, and well-proved solutions as a contrast to more specialized ad hoc solutions, which tend to be invented to deal with special purpose needs in narrow application domains.

Although beneficial, as argued above, programmatic approaches to problem solving among WWW authors, who deal with generated WWW documents, are usually not considered seriously. Rather, interactive approaches using special purpose tools with almost fixed functionalities dominate in the WWW authoring domain. The underlying problem is that only a minority of the WWW authors feel comfortable writing programs. This is, of course, a serious concern. Our attitude is, by and large, to ignore this problem. Consequently, we are providing tools only for a minority of WWW authors. The typical author using our approach is minded for programmatic problem solving. He or she, in addition, feels a need for more power than typically provided by the interactive tools, such as SGML/XML/HTML structure editors.

3 Examples of a programmatic authoring process

Before we continue the discussion at the general level we wish to give the reader a concrete feeling for the programmatic approach to WWW authoring that we use in our work. Thus, in this section we will illustrate the LAML approach to WWW authoring, which is based on the functional language Scheme from the Lisp family of programming languages together with a number of libraries. In subsequent sections of the paper we will broaden the discussion both with respect to WWW authoring using functional programming languages and with respect to the LAML fundamentals.

Let us assume that we want to develop a multiple-choice quiz service on the

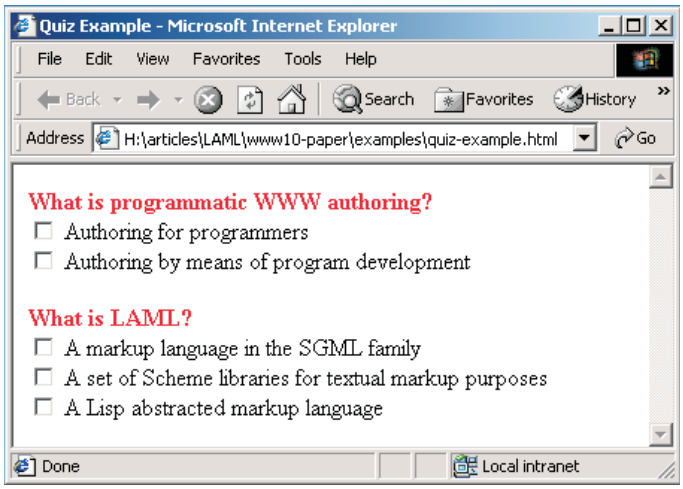

Figure 2: *A sample quiz.*

WWW. Figure 2 shows an example of a page with a couple of questions and possible answers. The user of the quiz service is supposed to select the right answers; Upon submission, the user will receive feedback, explaining why (or why not) the checked answers are correct. The quiz page as such is considered as a static WWW document. The answer page will be calculated (cf. the classification introduced in figure fig:inet-prog-categories) because it is created by the WWW server based on the choices made by the quiz reader.

From an authoring point of view it is important that the quiz can be formulated at an appropriate level of abstraction. We therefore form a tiny language which allows us to describe a quiz in terms of questions and answers. Figure 3 shows a sample quiz description, corresponding to the quiz shown in figure 2. A quiz consists of a number of quiz entries, each of which consists of a single question and a number of possible answers. Each answer has a formulation of the answer, a measure of correctness, and an answer clarification. The measure of correctness is a number between 0 and 100, where 0 means 'wrong', 100 means 'correct', and numbers in between represent partial correctness. The answer clarification is a description that explains *why* a given answer is wrong, partially correct, or correct.

A quiz description, like the one shown in figure 3, is an expression in the functional programming language Scheme. As one of the unique properties of Scheme (and, in general, of Lisp), a program fragment like the quiz expression in figure 3, is a list. The list concept constitutes the primary data structure of the programming language. As one particular nice implication of this, an expression like the quiz expression can be parsed by a generic Lisp parser (the Scheme reader) which is part of every Lisp system. Thus, we do not need any particular parser in order to process the document in figure 3.

When the top level quiz expression is evaluated by the Scheme interpreter a string is returned, which represents an underlying HTML presentation of the

```
(load (string-append laml-dir "laml.scm"))

(style "simple-quiz")

(quiz
  (list
    (quiz-entry
      (question-formulation "What is programmatic WWW authoring?")
      (answers
        (list
          (answer
            (answer-formulation "Authoring for programmers")
            (answer-correctness 0)
            (answer-clarification
              "Using programmatic authoring the document source is a program"))
          (answer
            (answer-formulation "Authoring by means of program development")
            (answer-correctness 100)
            (answer-clarification "The source of each WWW document is program")))))

    (quiz-entry
      (question-formulation "What is LAML?")
      (answers
        (list
          (answer
            (answer-formulation "A markup language in the SGML family")
            (answer-correctness 0)
            (answer-clarification "LAML is a markup language using Lisp syntax"))
          (answer
            (answer-formulation
              "A set of Scheme libraries for textual markup purposes")
            (answer-correctness 90)
            (answer-clarification
                "In addition there are various tools in the LAML package"))
          (answer
            (answer-formulation "A Lisp abstracted markup language")
            (answer-correctness 100)
            (answer-clarification
              "The name covers document styles, libraries and tools")))))))
```

Figure 3: *An LAML quiz description.*

quiz. In most situations it is convenient to have the Scheme interpreter return an HTML equivalent of the LAML expression. After all, the most typical action on our documents is to present them in a browser. In general, however, the quiz expression can be seen as a declaration of some basic quiz facts; The quiz expression can be evaluated, traversed, transformed or queried in many different ways, using special purpose interpreters. In that respect, the Scheme quiz expression is just like an XML fragment which uses a number of different quiz elements (XML tags).

In Scheme it is easy to program functions that transform the quiz expression

```
(quiz
   (filter
      black-or-white-quiz-entry?
      (list
         (quiz-entry
            ...)

         (quiz-entry
            ...)
      )
   )
)
```

Figure 4: *A sketch of a filtered LAML quiz description.*

```
(load (string-append laml-dir "laml.scm"))
(style "simple-quiz")

(let ((quiz-list (file-read "my-quiz.scm")))
 (quiz
  (filter
    black-or-white-quiz-entry?
    (map make-quiz-entry-from-list quiz-list))))
```

Figure 5: *A quiz read from a file and processed via higher-order functions.*

in figure 3 to HTML. We make use of the LAML mirror of HTML in which each HTML element (tag) is represented as a function in Scheme. The mirror will be discussed in more details in section 5. The functions involved in the transformation from the programmatic expression to HTML are shown in appendix A.

We will now illustrate the power and flexibility we obtain by representing the quiz as an expression in a functional programming language. Let us assume that we only want to deal with the subset of questions that exclusively contains correct or wrong answers. In other words, we wish to disregard the questions to which we present partially correct answers. Relative to the example in figure 3 this eliminates the last question. The change of the quiz can be realized by a simple filtering of the quiz questions, as sketched in figure 4. The `filter` function is a commonly used higher-order function which is supported in many functional programming languages. An application of `filter` returns the sublist of a list whose elements satisfy a predicate. The predicate `black-or-white-quiz-entry?` is found as part of the program definitions in appendix A.

The use of programmatic elements in the quiz document can be brought much further if the quiz author wants to. Figure 5 shows yet another version in which the quiz is taken from a file, which uses a simple list format, by means of

the function `file-read`. The quiz, as present on the file "`my-quiz.scm`", is first transformed somehow by mapping the function `make-quiz-entry-from-list` on all quiz elements. This produces a list of quiz entries that are filtered in the same way as described above.

When we are going to complete the Quiz service we are likely to write a CGI program on the WWW server which checks the answers. (In this example, we could as an alternative, carry out the check at the client side—typically using a Java Script program, but we will not deal with this alternative in the present paper). As the feedback from the WWW server, we chose to return the original quiz page with embedded qualitative evaluation of the answers. Using this approach, it is possible to submit the quiz once more, and hereby explore the quiz in additional details. In this context it is worth noticing that the functional paradigm fits nice as well. The server side program receives as input the choices made by the quiz user, and as result it returns a specialized version of the quiz page. Many simple server side solutions work in this way.

By using a programmatic authoring approach, as illustrated in figure 3 through 5, the gap between the quiz WWW page, as written by the quiz author, and the server side checking program becomes much smaller. Both parts of the systems can be programs in the same programming language, and the server side checking program can use some of the same functions as used by the quiz author, cf. appendix A.

Let us briefly discuss how the quiz example could be developed using a more conventional approach. It would not be attractive to author the quiz interface in pure HTML (using either an HTML editor or a text editor). The author would typically (but due to CSS ([1]), not necessarily) commit himself or herself to a particular presentation of the quiz which is only of minor importance to the quiz apparatus as such. Some of the functions in appendix A, such as `present-quiz-entry`, represent style sheet information. This could involve CSS expressions , but this is not the case nor important in our setup, because the documentation source already is separated from layout concerns. Thus, the programmatic authoring approach can be used to obtain many of the advantages known from the application of external and separate style sheets.

Many authors would probably be tempted to make a pure server side solution to the problem, taking the quiz input from some data structure or database, and generating HTML output via use of PHP, ASP, JSP or similar frameworks. This, however, involves mixed HTML work and programming in a conventional imperative style, which we find is less attractive than the functional style illustrated above. We will discuss this issue in more details in the following section of this paper.

As yet another alternative, some authors would probably be tempted to author the quiz in XML. This will involve the construction of a DTD (corresponding to context free grammar of the quiz language), quiz XML authoring quite similar to the quiz shown in figure 3, and programming of transformers from XML to HTML (using XSL, for instance). Finally, the server side checking must be done, in yet another framework (such as PHP, ASP, or Java). As can seen, this solution involves several, special-purpose technologies, whereas the Scheme/LAML approach only involves a single framework. We see this as

```
procedure print-quiz;
begin

  print-quiz-entry-start("What is programmatic WWW authoring?");
    print-answer("Authoring for programmers", 0,
                 "Using programmatic authoring the document source is a program");
    print-answer("Authoring by means of program development",
                  100, "The source of each WWW document is program");
  print-quiz-entry-end;

  print-quiz-entry-start("What is LAML?");
    print-answer("A markup language in the SGML family", 0,
                 "LAML is a markup language using Lisp syntax");
    print-answer("A set of Scheme libraries for textual markup purposes", 90,
                 "In addition there are various tools in the LAML package");
    print-answer("A Lisp abstracted markup language", 100,
                 "The name covers document styles, libraries and tools");
  print-quiz-entry-end;
end;

begin
  print-quiz;
end.
```

Figure 6: *A quiz document using the imperative programming paradigm.*

an important advantage of our programmatic authoring approach.

4 Paradigm and language issues

In this section we will discuss the use of the imperative programming paradigm for programmatic WWW authoring. As a subsequent discussion we will deal with some more detailed properties of programming languages, which are important when we use programming languages for WWW authoring purposes.

4.1 The imperative paradigm

We claim that the imperative programming paradigm does not fit well with the needs of the WWW author (nor, for that sake, for the server side programmer). The argumentation will be rooted in the example shown in figure 6, which is similar to the example in figure 3 of this paper.

Each of the procedures in the program shown in figure 6 prints text, say to standard output. The procedure called `print-answer` corresponds to the `answer` function of figure 3. We chose to split the function `quiz-entry` into `print-quiz-entry-start` and `print-quiz-entry-end`. It would be difficult to handle a separate `print-quiz-entry` procedure because it would need a large amount of parameters in the general case (all possible answers, percent numbers, and clarifications to a question).

The single most important thing to notice about the imperative solution is that procedure calls do not nest in the same way as function calls. The reason is that procedure calls are commands to which we can pass various kinds of parameters, none of which are commands themselves. As a contrast, function calls are expressions to which we can pass other function calls as actual parameters. The function calls in figure 3 nest in a way which is similar to the nesting of elements (tags) in an HTML or XML document. This nesting of elements is very difficult to mimic in an imperative solution. We could go for a mixed-paradigm solution in figure 6 (application of functions side by side with procedure calls), but this would blur the discussion.

The comparison from above also holds when we deal with CGI-like programming at the server side. In an ASP or PHP program, which uses imperative programming techniques, we see the following pattern again and again:

```
print(start-tag-1, attributes-1);
  print(contents-1)
  print(start-tag-2, attributes-2);
    print(contents-2);
  print(end-tag-2);
print(end-tag-1)
```

As a contrast, this corresponds to the function call

```
tag-1(
 attributes-1,
 concatenate(
   contents-1,
   tag-2(attributes-2, contents-2)))
```

or in Lisp syntax

```
(tag-1
  attributes-1
  (concatenate
    contents-1
    (tag-2 attributes 2 contents-2)))
```

in which an outer context (not shown) may handle the necessary file output issues.

It may be argued that the imperative solution is more efficient than the functional counterpart, both with respect to time and space. Using our approach we accumulate a potentially large text string which eventually is written to a file. During this process, a lot of temporary strings will be generated, which calls for subsequent garbage collection. Some of the complicated static documents that we generate from a programmatic source (such as LENO [10]) take a number of seconds to generate. Due to the huge memories of modern computers, however, we have never experienced any memory capacity problems during the generate process.

4.2 Programming language details

In this paper we go for radical solution to the WWW authoring problem in which the source of a WWW document is a program that fulfills the rules of a

programming language. This is not a problem for server side development, which is generally accepted to be within the programmatic domain. With respect to more conventional, static WWW authoring of a set of interlinked WWW pages, the case is different. The question is the following:

> *Can we in a reasonable way write a program which serves as the source of a set of interrelated WWW pages? The WWW author use the programming language and environment as an authoring environment. When the program is executed it is supposed to generate the underlying WWW pages, which can be accessed via the Internet.*

Programming languages with complicated syntaxes and rules are not good for this purposes. It would probably take an extraordinary enthusiastic C, C++, Pascal, or Java programmer to write a (non-server based) set of interrelated WWW pages directly in one of these programming languages.

Programming languages with a more flexible syntax and fewer rules are better candidates. We have found that Scheme can be used with success, and we believe that several other functional languages are candidates as well.

Next to the issue of a flexible syntactic composition of programs (and thus WWW documents) comes the question of *type safety*. Programming languages with static typing or type inference allow us to identify type errors before the program is executed. If we use a programming language with static typing for programmatic authoring we can ensure, as part of program compilation, that the resulting document is valid. However, the price for this safety can be relatively high in terms of rigid and limited composition and processing of programmatic documents. As a contrast, programming languages with dynamic typing deal with typed data objects at run time. Such languages cannot guaranty the documents to be valid, but they can provide for flexible document composition and processing, based for instance on generic list types, as known from Lisp.

The discussion above easily boils down to the contrast between a *theoretical approach* and a more *practical approach* to programmatic authoring. Using the theoretical approach, the checking of documents becomes the main focus of the research. Using the practical approach, the flexible creation of a complex set of documents (a few of which may be non-valid or problematic) is the most important concern. As it appears from the discussion above, we adhere to the practical approach in our LAML research and development work.

4.3 Mixing markup and programs

WWW authoring involves expressions in markup languages as a central element. If we use a programmatic approach we may end up using a mixture of programming language constructs and text with markup in a single document. Take as an example the case where all the HTML markup is located as strings in print commands, or where all program parts are located inside particular tags. This is aesthetically unpleasing, but even more important, it makes it difficult to use programmatic solutions (automations or abstraction) uniformly throughout the entire document.

We therefore find it important to come up with a solution which integrates the programmatic constructs with the markup expression. In the LAML system it has been our choice to embed the expressiveness of HTML in the programming language. We could also go in the other direction, namely to embed the programming language in the markup language. The Latte system [3] is an example of this, cf. section 6. As mentioned in the introduction, the XML language XEXPR [6] is another such example. The rationale behind our solution is to use the most powerful language as a host for the less powerful language. It has been relatively straightforward to embed HTML/XML in Scheme, whereas the realization of Scheme as part of an XML language will require a lot of work.

It is also relevant to mention the role of program errors, either revealed at compile time or run time. Due to the 'nature of programming' we have to deal with such anomalies in a programmatic authoring process. Notice, however, that some of these errors have natural counterparts in an authoring process which uses a markup language. Such errors are often first revealed at browse time, by surprising rendering of the document in a browser.

5 LAML

LAML [8; 9; 7] means *Lisp Abstracted Markup Language*. LAML consist of a number of libraries, tools, and document styles all of which can be loaded by most existing Scheme systems.

The foundation of LAML is the mirror of HTML in Scheme. The mirror creates a Scheme function for each different element of HTML. In that way we eliminate HTML entirely from the document source program, but we still have the full HTML expressiveness available as Scheme functions. The mirror has been produced by parsing the HTML document type definition (DTD) followed by the necessary information retrieval and automatic generation of all the necessary Scheme definitions. The HTML expression

```
<tag a1 "v1" a2 "v2" ...  am "vm"> contents </tag>
```

is mirrored to the Scheme function call

```
(tag 'a1 "v1" 'a2 "v2" ...  'am "vm" "contents")
```

Figure 7 shows an example of two identical documents in HTML and LAML. As it appears, each `tag` function implicitly concatenates all the content strings following the attribute list. An underscore character in between content strings suppresses white space.

The tag function distinguishes its parameters by means of types (whether symbols or strings) and by their mutual positions. Thus, the first thing to do in the body of a tag function is to collect the list of attributes and to form the concatenated content string. The mirror is aware of all possible attributes of an HTML tag together with their types. It is possible to get a warning if invalid attributes names or invalid attribute value types are being used. In principle, it would also be possible to check the validity of the generated HTML document during the LAML interpretation process. However, this is not supported yet.

```
<html>
   <head> <title> Quiz Example </title> </head>
   <body>
      <font color="#ff0000"> <b> What is LAML? </b> </font>
      <br>
      <input type="CHECKBOX" value="true" name="a-1-1">
      A markup language in the <em> SGML family</em>.
      <br>
      <input type="CHECKBOX" value="true" name="a-1-2">
      A set of <em> Scheme libraries </em> for textual markup purposes.
      <br>
      <input type="CHECKBOX" value="true" name="a-1-3">
      A  <em> Lisp abstracted markup language</em>.
   </body>
</html>
```

```
(html
  (head (title "Quiz Example"))
  (body
    (font 'color "#ff0000" (b "What is LAML?"))
    (br)
    (input 'type "CHECKBOX" 'value "true" 'name "a-1-1")
    "A markup language in the" (em "SGML family")_"."
    (br)
    (input 'type "CHECKBOX" 'value "true" 'name "a-1-2")
    "A set of" (em "Scheme libraries") "for textual markup purposes."
    (br)
    (input 'type "CHECKBOX" 'value "true" 'name "a-1-3")
    "A" (em "Lisp abstracted markup language") _ "."
  )
)
```

Figure 7: *A HTML document and an identical LAML document.*

The mirror illustrated above—called the surface mirror—spends a fair amount of time to 'sort out' its parameters. There is a more basic mirror available, which avoids this work. If efficiency is very important, it may be a good thing to avoid the surface mirror. If, however, LAML is used for static authoring purposes, the surface mirror should definitively be used.

It is worth noticing that there is only very little gained by writing an LAML program, like the fragment in figure 7 instead of the HTML document shown above it. However, the LAML document can easily be abstracted to a more pleasant level for the author. In order to provide for this, the author can use one of the predefined LAML document styles, or the author can write the necessary Scheme functions himself or herself. The second `quiz-entry` of figure 3 may serve as an example of such an abstraction of the document in figure 7 (although the `quiz-entry` in figure 3 contains additional correctness and clarification information.)

The most problematic concern, raised by people who have been interested

in a programmatic authoring approach using Scheme and LAML, is the need for passing relatively small strings to Scheme functions. Let us illustrate the problem with the following fragment

```
(tag "A Lisp abstracted markup language")
```

in which we may want to emphasize the substring "Lisp" in an `em` tag:

```
(tag "A" (em "Lisp") "abstracted markup language")
```

This involves splitting the string "A Lisp abstracted markup language" in three parts of which the middle part is to be embedded in the `em` function (the mirror of the `em` tag). It is awkward and error prone to make these changes manually. Our solution is to provide an editor command (in Emacs) which works on a selection of the string "Lisp". Given the selection, the editor command splits the string and it embeds the selected substring in the `em` function. There is, of course, also a function which makes the unnesting and string splicing. It may be argued that the resulting document becomes hard to read, because the content is fragmented and represented as many small strings. However, the total amount of markup using LAML is smaller than the markup overhead in HTML, mainly due to the use of end tags in HTML (compare the upper and lower parts of figure 7.) So in our opinion, it is a matter of taste which of the formats to prefer.

Besides the HTML mirror, the LAML software package consists of a number of document styles and tools. A LAML document style defines a Scheme-based language for a particular purpose. The most important document styles are the LENO lecture note style for teaching materials, the course home page style, and the manual style (for producing manual pages of Scheme libraries). Some of the educational document styles are described in a separate paper [10]. A LAML tool is a facility that processes a program or a document in a particular way. The SchemeDoc tool is able to extract documentation comments from a Scheme source file, and to pipe these through the manual document style for production of library documentation. In addition there are LAML-based XML/HTML/Scheme parsers and pretty printers in the LAML tool box.

There is a strong connection between LAML and Emacs. Emacs can help the LAML user in several ways. The most important is *single key document processing*. Via single key processing, we can save and execute a LAML document by hitting a single key in Emacs. The program execution usually generates an underlying set of HTML files. In the simple cases, the processing of `f.laml` derives the file `f.html`. The other kinds of Emacs functionality is template support of often used fragments, and the `embed` and `unembed` editor commands discussed above.

6 Related Work

The work described in this paper is strongly connected to the Scheme programming language. Because of its power and relative simplicity, Scheme has been used in several different contexts for WWW related work.

Latte is mixture of the Latex text formatting system and Scheme, at least at the conceptual level [3]. In Latte, the author uses a Latex-like markup style. Most interesting, however, Latte mirrors a language similar to Scheme in the markup framework. This means that it is possible to make programmatic contributions to a Latte document by writing Scheme definitions in a Latex syntax. In order for this to be successful, it has required work at both the lexical and syntactical level of the Latte language.

BRL allows the WWW author to activate Scheme in designated places of a document [5]. The places are identified with square brackets. The Scheme program fragments within the square brackets are executed on the WWW server, using a slightly non-standard Scheme semantics. Thus, a BRL document is a mix of HTML markup and Scheme program fragments. BRL is particularly strong with respect to access of a relational database on the server side. Also of interest for this paper, BRL has a solution to the 'string parameter passing problem' discussed in section 5 of this paper. In BRL you can write

```
(brl-function ] text with <em> HTML </em> markup
   [brl-stuff] more free text[)
```

As it appears, free "HTML text" appear to be embedded in ']' ... '[' quotes. As an alternative understanding, BRL details are surrounded by '[' ...']' quotes. It is a matter of taste and experience if you like this particular notation. In our work we have gone for a solution that avoids the mixing of HTML markup and programmatic expressions.

Scheme has been used in other WWW contexts as well. Queinnec demonstrates that continuations, which represents one of the more advanced concepts in Scheme, can be used by a programmatic author to form trails through a material [11]. Queinnec uses the LAML libraries as part of his work.

The functional programming community has also reflected an interest to make a bridge between functional programming and WWW authoring. In the paper "Haskell and XML: Generic Combinator or Type-Based Translation?" Wallace and Runciman discuss two different approaches to write XML applications in Haskell [12]. As already touched on earlier in this paper, the interests of Wallace and Runciman are oriented towards valid documents and type checking. This is a contrast to our work on LAML which strives for a more flexible use of a programming language without much emphasis on strong and static typing.

Much of the literature on server side programming is also relevant here. Both ASP, PHP, and JSP operates by embedding program fragments in special tags of an HTML document, which are interpreted at the server side. Thus, documents using these technologies mix expressions in the markup language and a programming language. We have already argued (see section 4.3) that this creates a number of problems. As a contrast, LAML uses pure programmatic expressions from the functional paradigm.

7 Conclusions

In this paper we have advocated a radical—and probably controversial—approach to WWW authoring. It is called 'a programmatic approach' because it is based

on the idea of representing a document source as a program, written in a programming language. Program execution processes the document somehow, typically giving an equivalent HTML document (or a set of HTML documents) as the result.

We have reported on our experience with the use of Scheme for programmatic WWW authoring. During the last two years the author has written almost all his WWW material using LAML and Scheme. The material have had several thousand hits each month. Educational material and interactive educational WWW services are typical examples.

We have found that a Lisp language is very good for our purpose. As demonstrated in section 5 of this paper, we have been able to approach the conventional HTM/XML means of expression in Scheme syntax. At a more general level we have found that the functional programming paradigm is well-suited for documents with markup elements that are embedded into each other.

Scheme does not represent state of the art in functional programming languages. Many functional programmers would probably prefer Haskell instead. In our opinion, use of Haskell or a similar language would imply focus on type systems and document validation. These are important issues, but work focused on these aspects does not necessarily create any documents in the end. We claim that Scheme is a pragmatically good choice if our goal is to produce high quality and real life WWW material via use of a programmatic approach.

We have argued that traditional imperative languages are bad choices for programmatic WWW authoring. This extends to server-side use of the imperative paradigm as well. We are confident that object-oriented languages also can be used with success. However, much depends on the flexibility of the concrete programming language in use.

The chief advantages of the programmatic authoring approach are

- Uniform use of a single language—a programming language—in the authoring process.
- The possibility to use abstraction on subdocuments, and to automate routine tasks whenever and where ever needed.
- The ease of transferring documents from the static WWW domain to the server-dynamic WWW domain (which—by nature is programmatic).

The most important deficiency is that the author needs programming skills.

LAML is available as free software from the LAML home page [7]

A A simple LAML quiz document style

In this appendix we show the simple quiz style, which has been developed to demonstrate LAML in this paper.

```
; LAML Library loading
(lib-load "file-read.scm")
(lib-load "html4.0-loose/basis.scm")
(lib-load "html4.0-loose/surface.scm")
(lib-load "html4.0-loose/convenience.scm")
```

```
(lib-load "time.scm")
(lib-load "color.scm")

; Return a function which tags some information with tag-symbol
(define (tag-information tag-symbol)
  (lambda information (cons tag-symbol information)))

; Tag function generation
(define quiz-entry (tag-information 'quiz-entry))
(define question-formulation (tag-information 'question-formulation))
(define answers (tag-information 'answers))
(define answer (tag-information 'answer))
(define answer-formulation (tag-information 'answer-formulation))
(define answer-correctness (tag-information 'answer-correctness))
(define answer-clarification (tag-information 'answer-clarification))

;;; Quiz entry selectors
(define question-of-entry (make-selector-function 2))
(define answers-of-entry (make-selector-function 3))

;;; Question selector
(define formulation-of-question (make-selector-function 2))

;;; Answer list selector
(define answer-list-of-answers (make-selector-function 2))

;;; Answer selectors
(define answer-formulation-of (compose second (make-selector-function 2)))
(define answer-correctness-of (compose second (make-selector-function 3)))
(define answer-clarification-of (compose second (make-selector-function 4)))

;; Present the quiz list q-list
(define (quiz q-lst)
 (write-text-file
  (let ((n (length q-lst)))
   (page
     "Quiz Example"
     (list-to-string
      (map2 present-quiz-entry q-lst (number-interval 1 n))
      (p))))
  (string-append (source-filename-without-extension) ".html")))

; Present a single quiz entry qe, which is assigned to the number n.
(define (present-quiz-entry qe n)
 (let* ((question (formulation-of-question (question-of-entry qe)))
        (answers (answer-list-of-answers (answers-of-entry qe)))
        (m (length answers)))
  (con (font-color red (b question)) (br)
       (list-to-string
         (map2
           (lambda (a m) (present-answer a n m))
           answers (number-interval 1 m))
         (br)))))

; Present a single answer a in quiz entry n.
; This answer is assigned to the the number m.
(define (present-answer a n m)
```

```
 (let ((formulation (answer-formulation-of a))
       (answer-id (make-id n m)))
   (con (checkbox answer-id #f)
        (horizontal-space 1)
        formulation)))

; Make an internal answer identification string based on two numbers.
(define (make-id n m)
  (string-append "a" "-" (as-string n) "-" (as-string m)))

; Has quiz-entry only correct or incorrect answering possibilities
(define (black-or-white-quiz-entry? quiz-entry)
  (let ((answer-list (answer-list-of-answers (answers-of-entry quiz-entry)))
        (partial-correct-answer?
          (lambda (answer)
            (let ((n (answer-correctness-of answer)))
              (and (> n 0) (< n 100)))))
       )
    (null?
      (filter partial-correct-answer? answer-list))))

; Make a quiz entry from a list lst
(define (make-quiz-entry-from-list lst)
  (let* ((question (first lst))
         (a-lst (second lst)))
    (quiz-entry
       (question-formulation question)
       (answers (map make-answer-from-list a-lst)))))

; Make an answer entry from a list lst
(define (make-answer-from-list lst)
  (let ((fo (first lst))
        (co (second lst))
        (cl (third lst)))
    (answer (answer-formulation fo) (answer-correctness co) (answer-clarification cl))))
```

References

[1] Bert Bos, Hkon Wium Lie, Chris Lilley, and Ian Jacobs. Cascading style sheets, level 2 css2 specification. Technical report, W3C, May 1998.

[2] Sharon Adler et al. Extensible stylesheet language (xsl) version 1.0. Technical report, W3C, November 2000.

[3] Bob Glickstein. Latte—the language for transforming text. Located on http://www.latte.org/, 1999.

[4] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[5] Bruce R. Lewis. Brl—a database-oriented language to embed in html and other markup. Located on http://brl.sourceforge.net/, October 2000.

[6] Thomas Nicol. Xexpr - a scripting language for xml. W3C note located at http://www.w3.org/TR/, November 2000.

[7] Kurt Nørmark. The LAML home page. http://www.cs.auc.dk/-~normark/laml/, 1999.

[8] Kurt Nørmark. Programming World Wide Web Pages in Scheme. *Sigplan Notices*, 34(12):37–46, December 1999. Also available via [7].

[9] Kurt Nørmark. Using Lisp as a markup language—the LAML approach. In *European Lisp User Group Meeting*. Franz Inc., 1999. Available via [7].

[10] Kurt Nørmark. A suite of www-based tools for advanced course management. In *Proceedings of the 5ht annual SIGCSE/SIGCU Conference on Innovation and Technology in Computer Science Education*, pages 65–68. ACM Press, July 2000. Also Available from http://www.cs.auc.dk/ normark/laml/.

[11] Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 23–33. ACM Press, September 2000.

[12] Malcolm Wallace and colin Runciman. Haskell and xml: Generic combinators or type-based traslation? In *Proceedings of the ACM SIGPLAN International Conference on functional programming*, pages 148–159, 1999. Published in Sigplan Notices vol 34 number 9.